UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Antonio Fajdiga

Aspektno usmerjen razvoj programske opreme in primerjava s tradicionalnimi razvojnimi pristopi

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: doc. dr. Damjan Vavpotič

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01959 / 2013
Datum: 2.9.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANTONIO FAJDIGA**

Naslov: **ASPEKTNO USMERJEN RAZVOJ PROGRAMSKE OPREME IN PRIMERJAVA S TRADICIONALNIMI RAZVOJNIMI PRISTOPI**
ASPECT ORIENTED SOFTWARE DEVELOPMENT AND COMPARISON WITH TRADITIONAL DEVELOPMENT APPROACHES

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V okviru diplomske naloge predstavite področje aspektno usmerjenega razvoja programske opreme in posebnosti, ki se pri takem razvoju pojavljajo pri postopkih zajema zahtev, načrtovanja in implementacije. Tak razvoj primerjajte s tradicionalnimi, še zlasti objektno-usmerjenimi razvojnimi pristopi. Predstavite tudi orodja, ki tak aspektno usmerjen razvoj podpirajo. V nalogi na kratkem primeru predstavite tudi uporabo aspektov v programski kodi.

Mentor:

doc. dr. Damjan Vavpotič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Izjava o avtorstvu diplomskega dela

Spodaj podpisani **Antonio Fajdiga**,
z vpisno številko **63070491**,
sem avtor diplomskega dela z naslovom:

Aspektno usmerjen razvoj programske opreme in primerjava s tradicionalnimi razvojnimi pristopi

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Damjana Vavpotiča
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne Podpis avtorja:

Zahvaljujem se mentorju prof. dr. Damjanu Vavpotiču za vso pomoč in napotke pri izdelavi diplomskega dela. Zahvaljujem se tudi staršem, ki so mi omogočili študij na Fakulteti za računalništvo in informatiko v Ljubljani.

Kazalo

Povzetek

V zadnjih nekaj desetletjih je računalniška tehnologija nedvomno povečala stopnjo izmenjave informacij. Tehnologija vsakodnevno napreduje in nam omogoča enostavno, hitro in udobno komunikacijo. Če pogledamo razvoj računalniške tehnologije v zadnjih letih, lahko ugotovimo, da se je predvsem na področju računalniške programske opreme zelo spremenila.
V diplomski nalogi smo se osredotočili na metodologije razvoja programske opreme, aspektno usmerjen razvoj programske opreme in na primerjavo z objektnim razvojem. V uvodnem delu smo obravnavali razvoj računalniške programske opreme skozi leta. V nadaljevanju smo govorili o metodologiji programske opreme, o tem, kaj metodologija sploh je, zakaj podjetja potrebujejo metodologijo in kakšno vrsto metodologije uporabljamo v določenih primerih. Sledila sta podrobnejša obravnava in pregled objektnega in aspektnega razvoja programske opreme. Ker je relativno nov pojem v računalništvu, smo aspektno usmerjen razvoj programske opreme postavili v središče pozornosti. V zadnjem poglavju smo naredili še primerjavo med objektnim in aspektnim razvojem programske opreme in na koncu prišli do zaključka, katera metodologija je primernejša v konkretnem primeru.

Ključne besede: metodologija, razvoj programske opreme, objektno usmerjen razvoj programske opreme, aspektno usmerjen razvoj programske opreme.

Abstract

Over the past few decades the advance of computer technology have undoubtedly increased the rate of information exchange. It progresses every day and thus communication across the globe is now done with ease, convinience, and speed. However, in tremendous contrast to its development years ago, it can be seen that computer technology, and moreover computer software development has changed a lot.

It is why in the thesis, that you have before you, we concentrates around the problem of computer software development methodology, aspect-oriented software development and comparation with the object-oriented software development. In the introductory part we addressed the development of the computer software throw the years. In the following we talked about the software methodology; what is methodology, why companies need a methodology at all and what kind of methodology we use in certain cases. In addition, we are more concentrated on object-oriented software development and aspect-oriented software development. As a relatively new term in computer science, the center of attention is set on the aspect-oriented programming and development. These two methodologies are compared in the final chapter, and from it we can achieve a conclusion as to which methodology is more useful in a certain matter.

Keywords: methodology, software development, object-oriented software development, aspect-oriented software development

Poglavje 1

Uvod

Doba računalniške tehnologije se je začela z nastopom štiridesetih let prejšnjega stoletja oziroma natančneje z esejem Alana Turinga " Computable numbers, with an application to the decision problem " [1], ki je postavil teorijo o programski opremi in njenih aplikacijah. Z drugimi besedami je pojem programska oprema uporabljen za opisovanje njenih aplikacij, vendar v računalniškem inženiringu nadomesti informacije, ki izvirajo iz računalniškega sistema, programov in podatkov. Organizirani pristopi za razvoj programske opreme so začeli nastajati s pojavom prvega računalniškega hrošča, leta 1946. Šestdeseta leta so torej prinesla veliko sprememb na področju računalniške tehnologije. Takrat, se je pojavila prva metodologija za razvoj sistemov. SDLC (angl. Software Development Life Cycle) lahko obravnavamo kot ogrodje prve metodologije za razvoj informacijskih sistemov. Glavna ideja je bil razvoj informacijskih sistemov na strukturen in metodičen način. Metodologija programske opreme je postopen način razvoja informacijskega sistema, ki vključuje uporabo različnih tehnik in orodij, celovit v smislu korakov življenjskega cikla razvoja. Sedemdeseta in osemdeseta leta, so bila kritična za računalničarje in inženirje. Takrat so se pojavile številne težave s programsko opremo, rešitev pa je bilo malo. Pri veliko projektih so šli prek svojih zmožnosti, in posledično so propadla številna podjetja. Programerji so se soočali s težavami, ki so zahtevale veliko časa. Nova formula se je obrestovala in pokazala po-

zitivne rezultate ter svetlejšo prihodnost. Za razvijalce programske opreme so bila sedemdeseta leta kaotična. Takrat je v ospredje prišlo strukturno programiranje, ki je znano kot programska paradigma, ki si prizadeva za izboljšanje jasnosti, kakovosti in časa razvijanja računalniškega programa. Strukturno programiranje, se je pojavilo leta 1969, po zaslugi dela Bohma, Jacopinija in Dijkstre [2], ki so skupaj prispevali pri Boehm-Jacopinjevem izreku, ki zagotavlja teoretične osnove tega programiranja. Sestavljen je iz treh načinov povezovanja, ki so: zaporedje, izbira in ponovitev. Navaja, da so ti trije načini dovolj za izražanje izračunljive funkcije. Zavedati pa se moramo, da Boehm-Jacopinjev izrek ne obravnava vprašanja, kako napisati program, ga analizirati in zagotoviti njegovo korist. To je bil skupni interes Dijkstre, Roberta W. Floyda, Tonya Haarea in Davida Griesa. Pozneje, v devetdesetih letih, se je začelo pomembno obdobje za računalniško tehnologijo, v smislu razvoja programske opreme. S prihodom objektnega programiranja, ki je bilo razvito v zgodnjih šestdesetih letih in je postalo prevladujoči pristop leta pozneje, je razvoj programske opreme postal lažji in bolj priljubljen kot kdaj koli prej. Pojme predstavlja kot objekte, ki imajo lastnosti in so sposobni opisati predmet, poleg tega pa se uporabljajo postopki, znani kot metode, objektno usmerjenega pristopa programiranja, ki so kmalu prišli v ospredje in postali močni. Jezika, kot sta Java in C++, sta odlična primera objektivno usmerjenih jezikov, ki obravnavajo objekte kot instanco razredov in se uporabljajo za medsebojno interakcijo, da bi ustvarili aplikacije in računalniške programe. Objektno usmerjen jezik, obravnava programe kot skupino predmetov, ki delujejo v nasprotju s strukturiranimi jeziki, pri čemer se program obravnava kot " seznam procedur ", ki jih je treba izpolniti v enakem vrstnem redu. Ko se je proti koncu devetdesetih let prvič pojavilo aspektno usmerjeno programiranje, ki je paradigma, katere cilj je povečati modularnost, ki omogoča ločitev med seboj presečih zadev (angl. cross-cutting concerns), je prineslo nov način razmišljanja in nov pogled k načrtovanju programske opreme.

Glavni cilj diplomske naloge je obravnava in podroben opis aspektnega ra-

zvoja programske opreme ter primerjava z objektnim razvojem programske opreme.

Poglavje 2

Metodologija razvoja programske opreme

Od začetka razvoja računalniških aplikacij do zgodnjih šestdesetih let so se aplikacije razvijale brez formalnih računalniških metodologij. V tem času je bil poudarek na samem programiranju in veščinah programerjev. Takrat so bili programerji in njihove veščine zelo cenjeni. Čeprav so bili zelo cenjeni, sami programerji niso posedovali vsega, kar je bilo potrebno za najboljši izkoristek teh veščin. Manjkale so jim dobre komunikacijske sposobnosti. Pogosto so se zanašali samo na svoje izkušnje in niso uporabljali neke formalne metodologije. Ker je samo ocenjevanje datuma, kdaj bo nek projekt končan v celoti, brez napak zelo težavno, je prihajalo do velikih zamud pri dostavi končnega izdelka. Programerji so bili preobremenjeni in so veliko časa porabili za odpravljanje napak v sistemu oziroma aplikaciji. Ponavadi so ljudje iz drugih oddelkov podjetja prihajali do programerja in ga prosili, naj jim naredi novo poročilo ali modifikacijo starega, ker so se aplikacije pogosto spreminjale. Te spremembe so predstavljale pravo težavo za uporabnika in tudi za same programerje. Poleg tega so spremembe vplivale še na celotni sistem in posledično je prihajalo do frustracij uporabnikov. Sčasoma je računalniška tehnologija vse bolj napredovala, s čimer se je izboljšalo tudi samo vodenje podjetja. Vse to je prispevalo k izboljšanju programske opreme in začetku

razvoja prvih metodologij.
Sam izraz metodologija razvoja programske opreme na nek način predstavlja ogrodje, ki se lahko uporablja pri strukturiranju, načrtovanju ter nadzoru razvojnega procesa. Metodologije razvoja programske opreme se ukvarjajo s procesom ustvarjanja programske opreme, ne toliko s tehnične strani kot pa z organizacijskega vidika.
Zgodovinsko gledano, so prvi procesi razvoja potekali brez formalnih metdologij. Šlo je za tako imenovane "ad hoc" metodologije [3]. Takšna metodologija lahko normalno deluje, le če imamo nek enostaven problem. Do želenega cilja lahko pridemo le, če uporabnik natančno ve kaj želi in če razvijalec ve, kako se to naredi prav in ima ustrezno opremo. Vendar pa je v večini primerov prihajalo do zamude ali pa končnega izdelka sploh nismo dobili.
Ko govorimo o metodologijah, je pomembno omeniti zakaj jih ljudje sploh uporabljajo in kakšne koristi imamo, če uporabimo neko metodologijo. V knjigi " Information systems development; methodologies, techniques & tools ", Avison in Fitzgerald [4] navajata tri ključne razloge, zakaj je dobro uporabljati neko metodologijo :

1. Boljši končni izdelek
2. Boljši razvojni proces
3. Standardizacija procesa

Sam razvoj metodologij, lahko razdelimo v več obdobij.

2.1 Predmetodološko obdobje

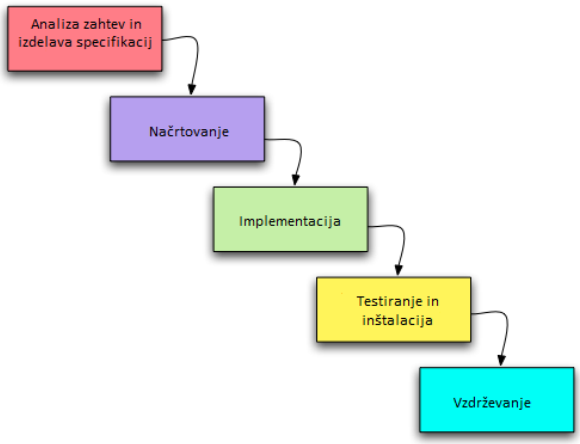
Kot smo omenili že na začetku tega poglavja, to obdobje traja od samega začetka razvoja programske opreme do šestdesetih let. V tem času je bilo veliko izkušenih programerjev, ki so bili primarno osredotočeni na razvoj programske opreme. Čeprav, so imeli znanje, niso posedovali organizacijske in komunikacijske sposobnosti. Čeprav te sposobnosti niso posedovali, se

je računalništvo zelo hitro razvijalo. Sčasoma so se podjetja spreminjala, pridobila so veliko organizacijskih veščin in vse to je na nek način vodilo do pojava prvih metodologij.

2.2 Pojav prvih metodologij

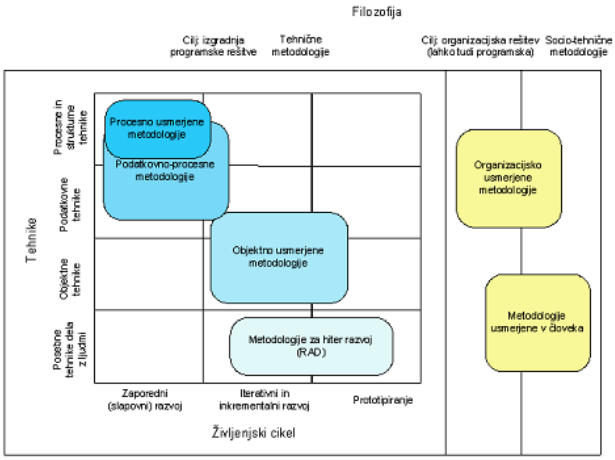
Pojem metodologija razvoja programske opreme se prvič omenja okrog leta 1960. Nekateri menijo, da lahko vzamemo SDLC [5] kot prvo formalizirano ogrodje za razvoj informacijskih sistemov. Glavna ideja SDLC-ja je razvoj informacijskih sistemov na premišljen, dobro strukturiran in metodičen način. Razvoj sistema ali aplikacije moramo razdeliti na faze, in sicer, od začetka ideje do dostave končnega izdelka. Vsako fazo, je treba posebej obravnavati. Pri razvoju programske opreme obstaja nekaj faz, ki jih srečamo pri uporabi metodologije SDLC. Te faze so prikazane na sliki 2.1.

1. Analiza zahtev in izdelava specifikacij
2. Načrtovanje
3. Implementacija
4. Testiranje in inštalacija
5. Vzdrževanje



Slika 2.1: Faze pri razvoju programske opreme [19].

Kot smo že omenili, lahko SDLC zgodovinsko gledano obravnavamo kot neke vrste prvo metodologijo, vendar pa sta se pozneje termin življenjski cikel in metodologija ločila. Pri sodobnih metodologijah lahko izbiramo različne življenjske cikle znotraj ene metodologije.
Danes lahko metodologije razdelimo glede na njihov tip, težo ter utežitev [6].



Slika 2.2: Delitev metodologij glede na njihov tip [6].

Delitev glede na njihov tip, lahko obravnavamo kot prvo delitev v metodologiji razvoja programske opreme.
Med socio-tehnične metodologije, lahko uvrstimo tudi organizacijsko usmerjene metodologije in metodologije usmerjene v človeka. Cilj teh metodologij ni nujna programska rešitev, temveč je lahko tudi samo organizacijska. V nadaljevanju so na kratko opisane vse metodologije, ki jih vidimo na sliki 2.2.

2.2.1 Procesno usmerjene metodologije

Metodologija je nastala v drugi polovici sedemdesetih let. Temelji na modeliranju procesov v sistemu. Za modeliranje procesov uporabljamo različne tehnike, kot so na primer: diagram podatkovnih tokov, odločitvena drevesa in tabele ter akcijski diagrami. Poleg tega uporabljamo tudi tehnike za prikaz razgradnje in strukture sistema, kot so strukturni diagrami. Procesno usmerjene metodologije uporabljajo slapovni življenjski cikel.

2.2.2 Podatkovno-procesne (mešane) metodologije

S temi metodologijami se prvič srečamo v začetku osemdesetih let. Podatkovno-procesne metodologije na nek način dopolnujejo procesno usmerjene metodologije, ker poleg samega modeliranja procesov, vsebujejo tudi modeliranje podatkov, ki nastopajo v sistemu. Glede tehnik pri podatkovno-procesnih metodologijah imamo tehnike za modeliranje procesov in tehnike za modeliranje podatkov.
Podatkovno-procesne metodologije uporabljajo slapovni življenjski cikel.

2.2.3 Objektno usmerjene metodologije

Pojem objektno usmerjena metodologija se prvič omenja v šestdesetih letih, a postane prevladujoča metodologija šele v devedesetih letih. Takrat se pojavijo prvi objektni programski jeziki, ki jih uporabljamo tudi danes.

Objektno usmerjene metodologije se bistveno razlikujejo od procesno in podatkovno-procesnih metodologij. Objektni pristop na nek način rešuje slabosti predhodnih metodologij. Procesne in podatkovno-procesne metodologije posebej obravnavajo statične in dinamične vidike sistema [7]. Tako se podatki in procesi modelirajo v različnih modelih. V fazi implementacije se podatki shranjujejo v podatkovni bazi, proces pa se kodira v programske procedure in rešitve, ki manipulirajo s temi podatki. Pri tem nastanejo težave pri spremembah podatkovne strukture, ki obvezno povzroči iskanje programskih modulov, ki uporabljajo spremenjeno strukturo. Zato prihaja do zamude in povečajo se stroški za razvoj sistema.
Objektno orientirana analiza in objektno orientirana konstrukcija omogočata boljše razumevanje problemov in boljše sodelovanje vseh zainteresiranih. Poveča se konsistentnost med analizo, načrtovanjem ter konstrukcijo in samim objektno orientiranim kodiranjem. Z analizo in načrtovanjem opišemo, kaj bomo delali, s konstrukcijo, kako bomo delali s kodiranjem pa implementiramo programsko aplikacijo. Objektni pristop omogoča lažje spreminjanje sistema, kar je posledica večkratne uporabnosti atributov in operacij ter tudi večkratne uporabnosti rezultatov analize, konstrukcije in kodiranja. Glede na klasične funkcionalno-strukturne pristope si od objektno orientiranega pristopa obetamo povečanje produktivnosti, povečanje kakovosti izvedenih projektov, poenostavitev vzdrževanja in poenostavitev nadgradnje. Objektni pristop pri razvoju informacijskih sistemov je pogosto povezan z inkrementalnim in iterativnim razvojem. Pri tem izdelke izdelujemo v več iteracijah, vsaka iteracija pa vzame izdelke iz predhodnih . Te izdelke le nadgradimo.

2.2.4 Metodologije za hiter razvoj

Te vrste metodologij so se pojavile kot odgovor na hitrejše spreminjanje zahtev. Do takrat razvite metodologije niso omogočale spremljanja hitrega razvoja programske opreme in sprememb zahtev. Zato je bila razvita metodologija, ki temelji na iterativnem življenjskem ciklu in prototipiranju.

2.2.5 Metodologije usmerjene v človeka

Metodologije so se pojavile kot kritika takratnih metodologij, ki so se osredotočale na tehnični vidik razvoja programske opreme. Sociološki vidik pa je bil zanemarjen. Pravijo, da moramo pri razvoju programske opreme upoštevati tudi sociološke komponente.
Lastnost teh metodologij je, da vključujejo vse tiste, na katere bo sistem vplival.

Poglavje 3

Aspektno usmerjen razvoj programske opreme

Razvoj programske opreme se sčasoma spreminja. Pojav in možnosti interneta, elektronskega poslovanja, eksponentno znižanje cen računalnikov in komunikacije ter povečana doba obstojnosti samega sistema močno pritiskajo na same razvijalce programske opreme, da čim prej naredijo in razvijejo nove sisteme. S tem prihaja tudi spodbuda za razvoj programskih procesov, ter za programiranje in zagotavljanje kakovosti. Za izdelavo vseh programskih sistemov, razen najbolj trivialnih, sta potrebna določeno inženirsko znanje in razdelitev sistemov na manjše dele, ki jih lahko lažje upravljamo.
V zadnjem desetletju dvajsetega stoletja, smo priča naraščanju, in tudi o prevladi objektno usmerjenega razvoja ter modularizaciji sistema. Objektno usmerjen razvoj se osredotoča na izbor objektov, kot na primarne enote modularizacije in samega obnašanja izbranih objektov sistema. Objekti so običajno elementi samega računalniškega procesa. Vendar pa je objektno usmerjen razvoj že dosegel svoje meje. Razvoj različnih sistemov istočasno zahteva manipulacijo z več zahtev. Odnos med zahtevami in komponentami sistema je zelo zapleten. Ena zahteva je lahko implementirana s strani več komponent in vsaka komponenta lahko vključi elemente več zahtev. V praksi to pomeni, da sprememba ene zahteve lahko vpliva na spremembo več kom-

ponent. Tudi če imamo priložnost ponovne uporabe navedenih komponent, to lahko pripelje do nezaželenega stroška, ker ponovna uporaba vključuje odstranitev dodatne kode, ki ni povezana z osnovnimi funkcionalnimi komponentami.

Aspektno usmerjeni programski razvoj (angl. Aspect-Oriented Software Engineering AOSE) je pristop za razvoj programske opreme, ki je namenjen reševanju prej omenjene težave in omogočanju lažjega vzdrževanja programov ter ponovni uporabi komponent. AOSE temelji na abstrakcijah, znanimi pod imenom aspekti, ki implementirajo sistemsko funkcionalnost, katera je lahko zahtevana na več različnih mestih v programu. Aspekti zajemajo funkcionalnost, ki je skupna več funkcijam, ki so vključene v sistem.

Ključna prednost uporabe aspektov je v tem, da podpirajo ločitev presečnih zadev. Zadeva je v našem primeru, vsaka pojmovna stvar, ki nas zanima. S predstavitvijo presečnih zadev kot aspekti, lahko le-te lažje razumemo, jih ponovno uporabimo ter spreminjamo neodvisno, ne da bi se obremenjevali, kje je ta koda uporabljena. Aspekti določajo ali bo neka presečna koda vključena pred ali po neki metodi.

Pri načrtovanju sistema, Jacobsen in Ng [21] predlagata, da je treba razmisliti o sistemu, ki podpira obnašanje različnih interesnih skupin, in je sestavljen iz osnovnega sistema ter razširitve.

Paketni diagram na sliki 3.1 prikazuje UML (angl. Unified Modeling Language) diagram za prikaz osnovnega sistema in razširitev. Osnovni sistem je nabor sistemskih funkcij, ki služijo izvajanju osnovnega namena sistema. Torej, če je namen določenega sistema ohraniti informacije o pacientih v neki bolnišnici, potem osnovni sistem omogoča ustvarjanje, urejanje, upravljanje in dostop do baze podatkov o evidenci pacienta. Razširitve osnovnega sistema predstavljajo dodatna obnašanja interesne skupine, ki jih je treba dodati osnovnemu sistemu. Na primer pomembno je, da zdravniški informacijski sistem ohranja zaupnost informacij o pacientu, tako da bi se ena razširitev v našem sistemu nanašala na nadzor nad dostopom teh informacij, druga pa bi se ukvarjala s šifriranjem teh podatkov.

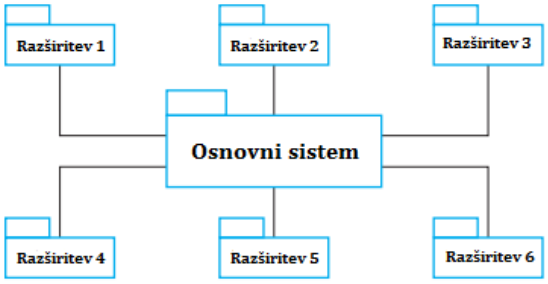
Obstajajo različne vrste razširitev, ki izhajajo iz različnih vrst obnašanja. V nadaljevanju so na kratko opisani.

1. Sekundarne funkcionalne razširitve. Dodajajo dodatno zmogljivost funkcionalnosti osnovnega sistema.

2. Razširitve v povezavi s sistemsko politiko. Dodajajo funkcionalno zmogljivost organizacijskih politik sistema. Razširitve, ki dodajajo varnostne funkcionalnosti, so primer teh razširitev.

3. Kakovostno storitvene(angl. Quality of Service - QoS) razširitve. Dodajajo funkcionalno zmogljivost, ki prispeva k uresničenju kakovosti zahtev za storitve, ki jih določa osnovni sistem. Primer takšnih razširitev je dodajanje predpomnilnika, ki lahko zmanjša število dostopov do baze podatkov.

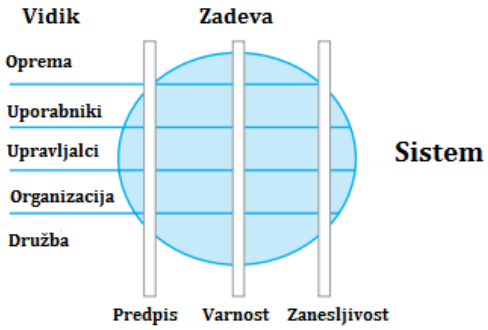
4. Infrastrukturne razširitve. Dodajajo funkcionalno zmogljivost za podporo uvajanja sistema za neko posebno izvedbeno platformo.



Slika 3.1: Osnovni sistem in razširitve [24].

Razširitve vedno dodajajo neko funkcionalnost ali dodatno karakteristiko osnovnega sistema. Aspekti predstavljajo en način uvajanja teh razširitev in jih je mogoče sestaviti skupaj z osnovnimi funkcijami osnovnega sistema z uporabo pravila tkanja v aspektno usmerjenem okolju.

Pojem ločevanja zadev je v svetu konstruiranja zahtev že dolgo znan. Vidiki, ki predstavljajo različne sistemske perspektive, so vključeni v številnih inženirskih metodah [23]. Le-te ločijo zadeve, ki so skupne različnim interesnim skupinam.
Vendar pa obstajajo tudi zahteve, ki presegajo vse vidike, kot je prikazano na sliki 3.2.
Da bi razvili sistem, ki je organiziran kot na sliki 3.1, moramo najprej identificirati zahteve osnovnega sistema ter zahteve za razširitev sistema. En način za ločevanje osnovne in sekundarne zadeve je, da vsak vidik predstavlja zahtevo ene interesne skupine. Če organiziramo zahteve po vidikih interesne skupine, lahko analiziramo in odkrijemo zahteve, ki se pojavijo v vseh ali večini vidikov. Ti vidiki so osnovne funkcionalnosti sistema. Ostale zahteve lahko implementiramo kot razširitev osnovnega sistema.



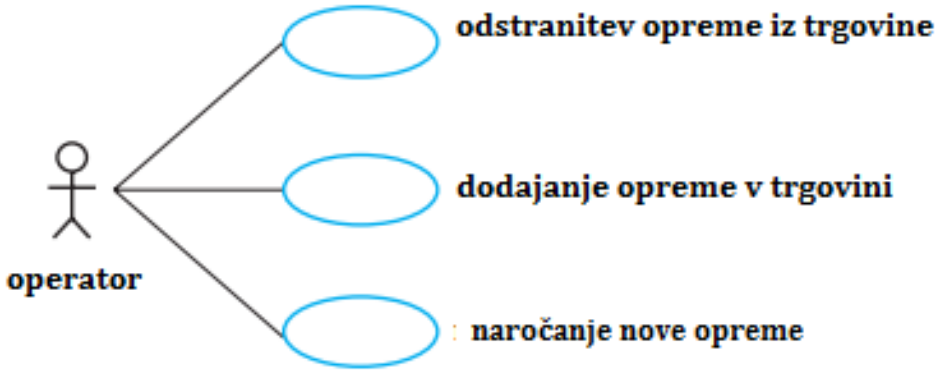
Slika 3.2: Vidiki in zadeve [24].

3.1 Aspektno usmerjeno načrtovanje in modeliranje

Aspektno usmerjeno načrtovanje je proces oblikovanja sistema, ki omogoča uporabo aspektov za izvajanje presečnih zadev in razširitev, ki so opredeljene

pri procesu konstruiranja zahtev. V tej fazi je treba analizirati zadeve, ki se nanašajo na problem, katerega moremo rešiti v ustrezne aspekte. Poleg tega moramo razumeti, kako se bodo ti aspekti združili z drugimi komponentami sistema in zagotoviti, da ne bo prišlo do nejasnosti glede združitev teh komponent.

Visoka raven samih zahtev zagotavlja osnovo za določitev nekaterih razširitev sistema, ki jih lahko izvedemo kot aspekt. Eden od načinov za predstavitev sistema in njegovih razširitev je raba primerov uporabe (angl. use cases). Primeri uporabe so interaktivno osredotočeni in podrobnejši kot zahteve uporabnikov. Predstavljajo most med zahtevo in načrtovanjem. V modelu primerov uporabe opišemo vsako interakcijo z uporabnikom in začnemo z identifikacijo ter opredelitvijo razredov v sistemu.



Slika 3.3: Primer uporabe za primer trgovine [24].

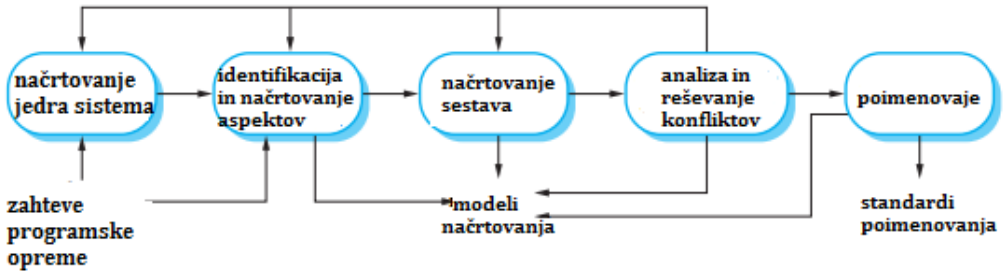
Jacobsen in Ng [21] opisujeta kako lahko primere uporabe uporabimo pri aspektnem načinu razvoja programske opreme. Predlagata, da je vsak primer uporabe predstavljen kot aspekt ter da razširimo model primera uporabe tako, da dodamo še podporo za prikaz stičišč (angl. join point) in stičnih presekov (angl. pointcut). Prav tako predstavita nov način prikaza razredov v primeru uporabe. Nov način vključi tudi prikaz fragmentov nekega razreda. Fragmente lahko na koncu spet sestavimo v celoto.

Slika 3.3 prikazuje tri primere uporabe, ki so lahko del sistema za upravljanje zalog. Ti prikazujejo odstranjevanje in dodajanje opreme v trgovini ter naročanje nove opreme. Dodajanje in naročanje opreme sta povezani med seboj. Ko nekaj naročimo, moramo te iste zadeve dodati v zalogo in jih dostaviti v eno izmed trgovin.

UML diagram že vključuje te razširitve primerov uporabe. Uporaba razširitev primera uporabe razširi funkcionalnost drugega primera. Slika 3.4 prikaže, kako naročanje opreme razširi osnovni primer uporabe za dodajanje opreme v določeni trgovini. Če oprema, ki jo je potrebno dodati v trgovino, trenutno ne obstaja, lahko naročimo novo opremo in jo dodamo v točno določeno trgovino. Presečne zadeve pa lahko predstavimo kot razširitev primerov uporabe. Jacobsen in Ng [21] predlagata, kako lahko implementiramo te vrste razširitev kot aspekte.

Slika 3.4: Razširitev primera uporabe [24].

Razvoj učinkovitega procesa za predstavitev aspektno usmerjenega načrtovanja je bistvenega pomena le, če je aspektni način načrtovanja sprejet in se uporablja. Aspektni način načrtovanja vključuje vse aktivnosti, prikazane na sliki 3.5. Te aktivnosti so:



Slika 3.5: Aspektno usmerjeni proces načrtovanja [24].

1. Načrtovanje jedra sistema. V tej fazi načrtujemo arhitekturo sistema za podporo osnovne funkcionalnosti sistema.

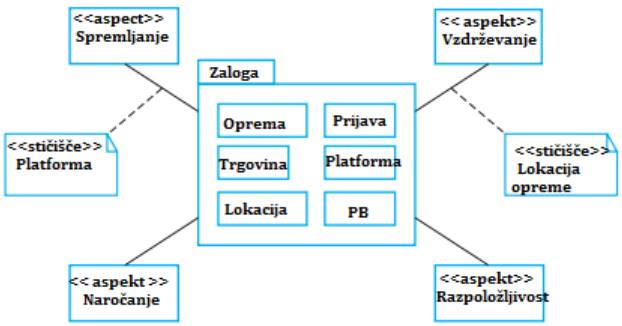
2. Identifikacija in načrtovanje aspektov. Začnemo z identifikacijo razširitve in analiziramo, ali se lahko aspekt razčleni na več podaspektov. Ko

identificiramo vse aspekte, potem lahko nadaljujemo z načrtovanjem vsakega aspekta posebej.

3. Načrtovanje sestava. V tej fazi analiziramo osnovni sistem in aspekte, da bi odkrili, kje je najboljše vstaviti te aspekte v osnovni sistem ne da bi porušili njegovo arhitekturo. V bistvu iščemo stičišča v programu, v katerih bomo tkali svoje aspekte.

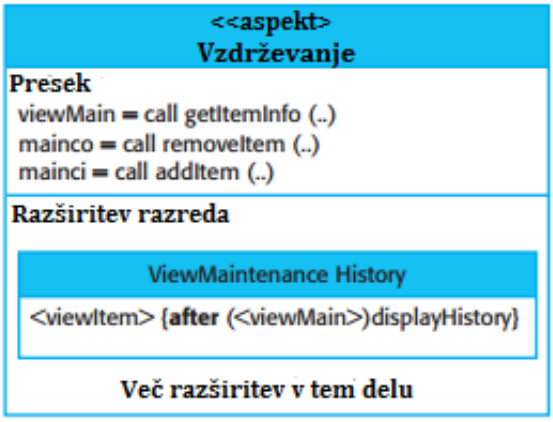
4. Analiza in reševanje konfliktov. Težava z aspekti je v tem, da lahko vplivajo drug na drugega, ko so vključeni v osnovni sistem. Konflikt se pojavi, ko imamo trk preseka različnih aspektov, ki določajo, da morajo biti vključeni na isti točki v programu. Obstajajo pa tudi prikriti konflikti, ko aspekte načrtujemo posebej in zato pride do spremembe osnovnega sistema, ki vpliva na strukturo aspektov. Prav tako lahko sprememba enega aspekta vpliva na ostale aspekte v sistemu.

5. Poimenovanje. Je pomembna aktivnost načrtovanja, ki določa standarde za poimenovanje entitete v programu. To je bistveno za preprečitev težav naključnih presekov. To se zgodi, če ima v nekem programu stičišče isto ime kot vzorec preseka. Zaradi tega se navodilo (angl. advice) veže na točko in ne na presek, kar privede do nepričakovanega vedenja programa.



Slika 3.6: Aspektno usmerjeni model načrtovanja [24].

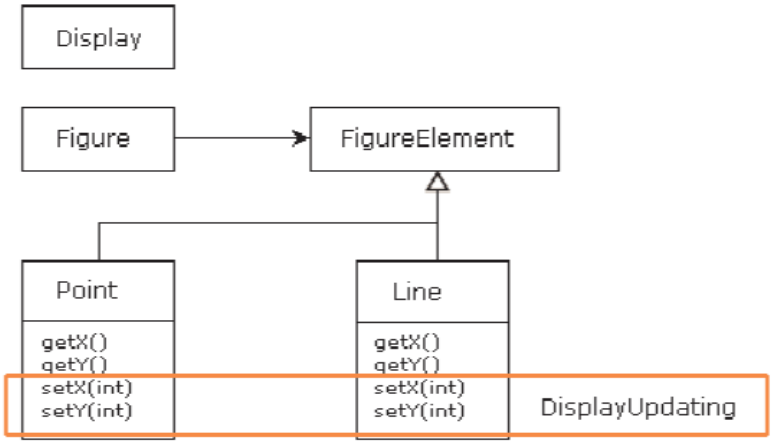
Tako kot pri objektnem imamo tudi pri aspektnem načinu razvijanja programske opreme iterativni razvoj, v katerem najprej nekaj razvijemo, nato analiziramo in ugotovimo, kje se da sistem izboljšati, na koncu pa naredimo novo iteracijo.
Rezultat aspektno usmerjenega procesa načrtovanja je aspektni model načrtovanja. Ta model predstavlja razširitev jezika UML, ki vključuje nove, aspektno usmerjene konstrukte, ki jih predlagajo Clark in Baniassad [22] ter Jacobsen in Ng [21]. Bistveni elementi aspektnega UML-ja so stičišča in navodila (angl. advices). Na sliki 3.6 je predstavljen aspektno usmerjen model načrtovanja. Uporabili smo UML diagram za predstavitev aspektov, ki sta ga predlagala Jacobsen in Ng. Prikazan je osnovni sistem in razširitve sistema (v našem primeru so to aspekti), ki jih sestavimo skupaj z osnovnim sistemom.
Slika 3.7 predstavlja podrobnejši model enega aspekta. Prikazan je aspekt " Vzdrževanje ". Preden začnemo načrtovati nek aspekta, moramo najprej narediti načrt osnovnega sistema. V nadaljevanju bomo pokazali, kako razvijemo aspekte s pomočjo aspektnega programiranja.



Slika 3.7: Del aspektnega modela [24].

3.2 Aspektno usmerjeno programiranje (angl. Aspect oriented programming - AOP)

Motivacija za pojav aspektno usmerjenega programskega pristopa, izhaja iz problemov, ki jih povzroča zapletenost in razmetanost kode. Kot rezultat tega, imamo presečne kode (angl. cross-cutting code) [8]. To je koda, ki je skupna več različnim razredoma in metodam. Problemov, povezanih s presečno kodo, ne moremo modularizirati z mehanizmi za dekompozicijo, saj zanje obstajajo različna nezdružljiva pravila.
Do problemov, povezanih z zapletenostjo kode, pride, ko lahko moduli v programskem sistemu sočasno vstopajo v interakcijo z različnimi zadevami (angl. concerns). Na sliki 3.8 je prikazan primer presečne kode.



Slika 3.8: Primer presečne kode [9].

Primer zapletenosti kode:

```
public void TransferTo(Account otherAcct, int amount) throws Exception
{
        Logger.BeginLog("Attempting transfer...");
        Security.VerifyAuthentication(); //throws exception if unauthrized
```

```
        Logger.Log("...authenticated...");
        if(balance<amount)
        {
                Logger.EndLog("...insufficient funds.");
                Notifier.DisplayError("Insufficient Funds");
                return;
        }
        otherAcct.Deposit(amount);
        balance−=amount;
        Logger.EndLog("...transfer successful.");
}
```

Vzemimo zgornji primer. Klic metode razreda Logger predstavlja implementacijo zadev, povezanih s postopkom prijave, medtem ko je klic metode VerifyAuthentication implementacija zadeve, ki skrbi za varnost sistema. Glavna metoda TransferTo pa je povezana z vodenjem računa. Na tak način so zadeve, povezane z varnostjo in postopkom prijave, pomešane z metodo za vodenje računa. To zapletanje predstavlja glavno oviro za enostaven razvoj in vzdrževanje programske kode. Do problemov, povezanih z razmetanostjo kode, pride, če je koda razširjena po več modulih.
Tako razvijalci pogosto sočasno razmišljajo o poslovni logiki, učinkovitosti, sinhronizaciji, beleženju dogodkov in varnosti. Takšna mnogoternost zahtev se odraža v sočasni prisotnosti elementov za implementacije posameznih zadev in zato postane koda prepletena.
Zapletenost in razmetanost kode gresta pogosto skupaj, čeprav govorimo o različnem konceptu.
Zapletenost in razmetanost kode vplivata na načrtovanje in razvoj programov na več načinov [8] :

1. Slaba sledljivost: Sočasna implementacija več zadev zakriva skladnost med neko zadevo in njeno implementacijo.

2. Nižja produktivnost: produktivnost zniža sočasna implementacija več zadev, ki preusmeri razvijalčevo pozornost od glavne zadeve na postranske.

3. Manj kode je ponovno uporabne: pod temi pogoji en sam modul imple-

mentira več zadev, zato se lahko zgodi, da drugi sistemi, ki zahtevajo podrobno funkcionalnost, niso sposobni takoj uporabiti takšnega modula, kar pa ponovno zniža produktivnost.

4. Slaba kvaliteta kode: koda s skritimi problemi povzroči nepreglednost. Sočasno mešanje več zadev lahko pomeni, da se eni ali več zadevam premalo posvetimo.

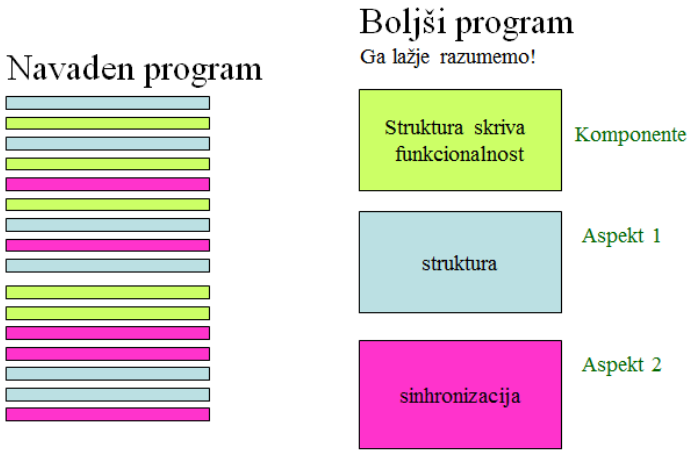
5. Težji razvoj: do načrta, v katerem so obdelane le nekatere zadeve, pogosto vodijo omejen pogled in omejeni viri. Obdelava bodočih zadev zahteva predelavo implementacije. Ker implementacija ni modularizirana, to pomeni, da moramo zadeve razdeliti na več modulov. Sprememba vsakega podsistema vodi do nekonsistence.

Veliko sistemov vsebuje presečne zadeve. Primeri takšnih zadev so [9] :

1. sinhronizacija
2. odkrivanje in odprava napak
3. upravljanje z varnostjo
4. upravljanje z pomnilnikom
5. sledenje in iztek seje

Zato se pojavljajo tehnike za modularizacijo. Ena od njih je aspektni način programiranja, ki je opisan v nadaljevanju.
Za aspektno usmerjen razvoj programske opreme se uporablja jezik, ki predstavlja razširitev običajnih programskih jezikov, ki omogočajo aspektno usmerjeno programiranje. Pojem AOP se je prvič pojavil leta 1997, ko je skupina ljudi iz podjetja Xerox Palo Alto Research Center predstavila svoj znanstveni prispevek na konferenci European Conference on Object-Oriented Programming (ECOOP) [10]. Naslov tega prispevka je " Aspektno orientirano programiranje " (angl. Aspect-Oriented Programming). Ta dokument formalno

zajema, kaj je aspekt in zakaj je pisanje kode, ki vsebuje presečne zadeve, brez uporabe aspektov zelo težko. V AOP-ju so presečne zadeve implementirane v aspektih, namesto da bi jih vgradili v osnovne module. Aspekti so na nek način enote modularnosti.



Slika 3.9: Prerez komponent in aspektov [8].

Slika 3.9 prikazuje, kako je sestavljen navaden program in kako je sestavljen program, ko si pomagamo z aspektnim načinom. Prednost pri uporabi aspektov je v njihovi ponovni uporabi. Aspekti so osnovni gradniki v arhitekturi z zadevami. Pogosto imamo podaspekte (angl. subaspects), kadar eni zadevi ustreza več kot en aspekt. Ker so aspekti zelo kompleksni, moramo imeti eksplicitne relacije med samimi aspekti sistema. Pri AOP poznamo tri razvojne korake, ki so prikazani na sliki 3.10 [8] :

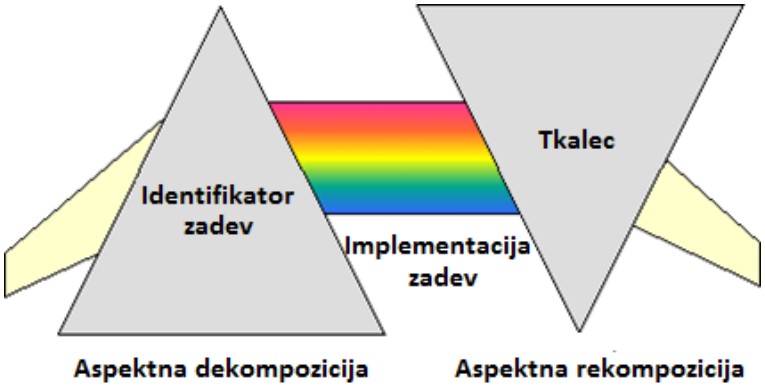
1. Aspektna dekompozicija

   V tem koraku razgradimo zahteve, da bi ugotovili preseke in skupne zadeve. Tukaj so ključne zadeve ločene od nivojskih zadev samega sistema.

2. Implementacija zadev

   V tem koraku je vsaka zadeva ločeno implementirana.

3. Aspektna rekompozicija
   V zadnjem koraku aspektni integrator opredeli rekompozicijska pravila, tako da ustvari modulizacijske enote, ki so znane pod imenom aspekti.



Slika 3.10: Razvojni koraki AOP [8].

Osnovni konstrukti jezika, ki jih uporablja AOP za definiranje prereznih zadev, so:

1. Stičišče (angl. join point): predstavlja neko točko v programu, kjer se nekaj zgodi, kot na primer: klic metode ali pa dodelitev neke spremenljivke, v katero imamo vmeščeno poslovno logiko. Stične točke so zelo pomembne, ker predstavljajo mesto, kjer je obnašanje aspekta tkano v aplikaciji. Poznamo statična in dinamična stičišča. Dinamična stičišča so natančno določene točke v izvajanju programa. Statična stičišča pa so lokacije v samem programu, kjer lahko dodamo nove člane. Dinamična lahko razdelimo v nekaj kategorij, in sicer: izvedbena stičišča, klicna stičišča ter stičišča, ki jih srečamo, ko imamo dostop do nekega polja. V tabeli 1 je prikazan podrobnejši opis teh stičišč.

Kategorija stičišč	Vrsta stičišč
Izvedbeno	Izvedba metode
	Inicializator izvedbe
	Konstruktor izvedbe
	Statični inicializator izvedbe
	Nadzor izvedbe
	Objektni inicializator
Klicno	Klic metode
	Klic konstruktorja
	Objektna pred-inicializacija
Dostop do polja	Sklicevanje polja
	Razporeditev polja

Tabela 3.1: Tabela stičišč

2. Stični presek (angl. pointcut): določa stično točko v programu, kjer imamo nek vidik (angl. concern). Stične preseke opredelimo tako, da imamo na levi strani ime preseka in njegove parametre, nato sledi dvopičje in na desni strani sam stični presek.

```
pointcut name(args):pointcut_designators;
```

Primer:

```
pointcut setter():call(void setX(int));
```

Presek se imenuje setter, sam presek pa call (void setX(int)).
Preseke lahko imenujemo glede na tip:

(a) execution(int *())
izbere izvajanje katere koli metode brez parametrov in vrne vrednost tipa int.

(b) call(* setY(long))
izbere klic katere koli metode setY, ki ima parameter tipa long.

(c) call(* Point.setY(int))
izbere klic katere koli metode setY razreda Point, ki ima vrednost int kot argument, ne glede na tip, ki ga vrača.

(d) call(* .new(int,int))
izbere kateri koli konstruktor razreda,vse dokler potrebuje natančno dve vrednosti int kot argument.

Pri presekih je dovoljena uporaba kompozicije. Preseke povežemo s pomočjo operetorja za negacijo (angl. not), in (angl. and) ter ali (angl. or). Na primer:

```
call(* *(..)) && (within(Line) || within(Point))
```

Stične preseke lahko razdelimo v tri kategorije, in sicer:

(a) ki temelijo na imenu,

(b) ki temelijo na lastnosti,

(c) Cflow

3. Navodilo (angl. advice): vedenje, ki se izvaja v neki stični točki. Izvrši se, ko dosežemo nek stični presek. Navodila so podobna metodam [8]. Programski jezik (AspectJ 5), ki ga bomo omenili v nadaljevanju, opredeljuje več vrst navodil:

(a) prednavodilo (angl. before advice): to navodilo se izvede, ko dosežemo neko stično točko, preden nadaljujemo s to stično točko. Primer:

```
before(param):pointcut(param) {
```

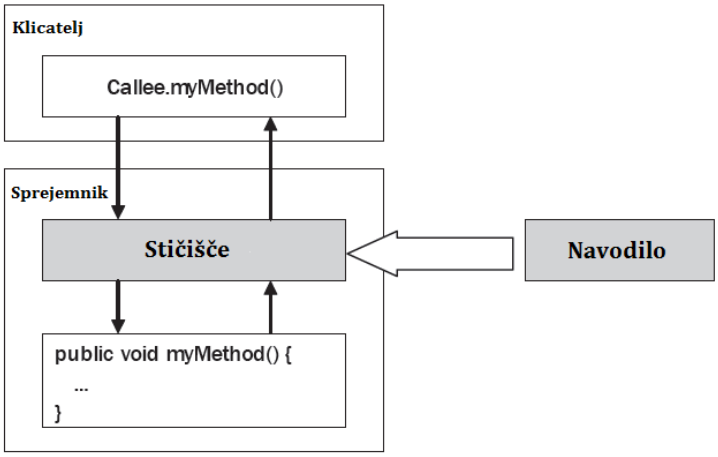
```
        body
}
```

(b) ponavodilo (angl. after advice): pri določeni stični točki izvedemo to navodilo potem, ko program nadaljuje s to stično točko. Primer:

```
after():set() {
Display.update();
}
```

(c) okoli-navodilo (angl. around advice): navodilo se izvede, ko dosežemo neko stično točko in nato program nadaljuje

4. Uvod (angl. introduction): ukaz, ki lahko naredi statične spremembe komponent aplikacije. Lahko na primer doda neko metodo razreda aplikacije.



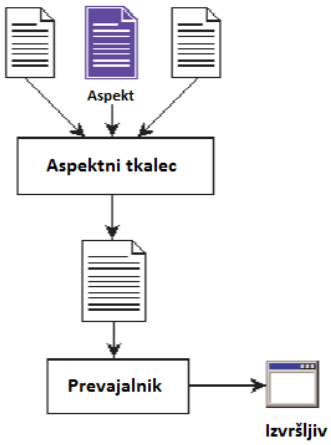
Slika 3.11: Anatomija AOP-ja.

Aspekt (angl. aspect) je pojem, podoben razredu. Skupaj s stičnimi preseki in navodili oblikuje presečno kodo. Slika 3.11 prikazuje anatomijo aspektnega programiranja.

3.2.1 Tkanje (angl. Weaving)

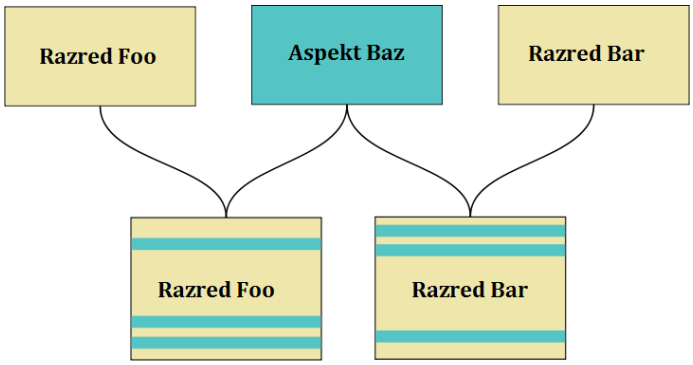
Ko enkrat opredelimo aspekte, je njihova uporaba določena v pravilih za izgradnjo. Ta pravila so na nek način vhod pomožnih programov, ki so znani pod imenom tkalec.
Aspektni tkalec je metaprogramski pripomoček za aspektno usmerjene jezike, ki je namenjen sprejemu ukazov določenih aspektov in ustvarjanju končne implementacijske kode. Tkalec deluje tako, da vzame ukaze, znane pod imenom navodila, in jih samodejno porazdeli po različnih razredih programa. Rezultat takega tkanja je množica razredov z imeni, enakimi, kot pri izvirnih razredih, vendar z dodatno oznako, ki jo dodamo v funkciji tega razreda. Nasvet določa točno lokacijo in funkcionalnost vstavljene kode. Skozi proces tkanja, ki je prikazan na sliki 3.12, aspektni tkalci vključijo kodo, ki bi bila sicer podvojena v razredih. Z odpravo tega podvajanja tkalci spodbujajo modularnost presečnih zadev. Aspekti definirajo implementacijo kode, ki bi bila sicer podvojena, ter uporabljajo stične preseke in stičišča, da bi opredelili navodila. Med procesom tkanja tkalec uporabi stične preseke in stičišča za ugotavljanje pozicije v kandidatnih razredih, pri katerih je treba dodati takšno implementacijo. Implementacija se potem doda v razredih na ugotovljenih točkah, pri čemer se koda izvede v ustreznem času brez zanašanja na ročno podvajanje programerja. Pri procesu tkanja obstaja več strategij, in sicer [11] :

1. poseben predprocesor, ki preveri izvorno kodo, ki se izvede med prevajanjem;
2. poprocesor, ki spremeni binarne datoteke;
3. AOP-prilagojen prevajalnik, ki generira tkane binarne datoteke;
4. LTW (angl. Load-Time Weaving), na primer pri Javi, ki naloži potrebne razrede v Javinem navideznem stroju (angl. Java Virtual Machine – JVM);
5. RTW(angl. run-time weaving)



Slika 3.12: Proces tkanja [9].

Večina jih podpira tudi CTW (angl. Compile-Time Weaving) z uporabo ene izmed prvih treh možnosti. Na primer, v Javi prevajalnik generira standardne binarne datoteke, ki jih izvrši vsak JVM. Datoteke .class se modificirajo glede na aspekte, ki jih definirajo. LTW ponuja fleksibilnejšo rešitev, ki potrebuje JVM classloader. LTW procesira Javino binarno kodo med izvajanjem in ustvari podatkovne strukture, kar je lahko počasno. Ko se naložijo, LTW nima vpliva na hitrost izvajanja.
Na sliki 3.13 je prikazan primer tkanja.



Slika 3.13: Primer tkanja [20].

Kot vidimo iz zgornje slike, aspektni tkalec vzame informacije iz razredov in ustvari nova razreda, ki vsebujejo aspektno kodo, tkano v samih razredih.

```
aspect Logger {
   pointcut method() : execution(* *(..));
   before() : method() {
      System.out.println("Entering " +
         thisJoinPoint.getSignature().toString());
   }
   after() : method() {
      System.out.println("Leaving " +
         thisJoinPoint.getSignature().toString());
   }
}
public class Foo {
   public void bar() {
      System.out.println("Executing Foo.bar()");
   }
   public void baz() {
      System.out.println("Executing Foo.baz()");
   }
}
```

Najprej definiramo en presek, ki se imenuje method() in potem napišemo še dve navodili befor() ter after(). Ko poženemo zgornjo kodo dobimo

```
public class Foo {
   public void bar() {
      System.out.println("Entering Foo.bar()");
      System.out.println("Executing Foo.bar()");
      System.out.println("Leaving Foo.bar()");
   }
   public void baz() {
      System.out.println("Entering Foo.baz()");
      System.out.println("Executing Foo.baz()");
      System.out.println("Leaving Foo.baz()");
   }
}
```

Zgornja koda je koda, ki jo je ustvaril sam tkalec.

3.2.2 Prednosti in slabosti uporabe AOP-ja

Ker AOP pomaga programerjem lažje ločiti zadeve in premagati težave, povezane s presečnimi zadevami, so koristi oziroma prednosti AOP-ja enake prednostim, ki izhajajo iz sposobnosti za modularizacijo implementacije presečnih zahtev. AOP pomaga pri premagovanju težav z razmetanostjo in zapletenostjo kode. Kot smo že omenili, vodita zapletenost in razmetanost kode do nekaterih neželenih posledic, kot so: nižja produktivnost, slaba sledljivost, slaba kvaliteta kode in težji razvoj. AOP obravnava vsako zadevo posebej z minimalno sklopljenostjo, ki ima za posledico modularno implementacijo, celo v primeru prisotnosti drugih presečnih zadev. Takšna implementacija naredi sistem, ki bo imel manj podvojene kode. Modularnost implementacije pomeni tudi obstoj sistema, ki je lažji za upravljanje in bolj razumljiv. Ker se aspektni moduli zavedajo obstoja drugih presečnih zahtev, je zelo enostavno dodati novejše funkcionalnosti z ustvarjanjem novih aspektov. Modularnost enako omogoča, da je koda ponovno uporabljiva.

Glede slabosti pa lahko omenimo, da je tehnologija AOP relativno nov pojem v računalništvu, ki se ne uporablja zelo pogosto, saj ni popolnoma testirana in dobro dokumentirana [9]. AOP je danes dobro opisana le v teoriji; tako da ni garancije, da bi ta tehnologija enako dobro funkcionirala v praksi kot v teoriji [9]. Še ena slabost tehnologije AOP je, da imamo omejeno število razvojnih orodij. AspectJ 5 je trenutno vodilna tehnološka razširitev programskega jezika Java za implementacijo tehnologije AOP. Vse to predstavlja težavo pri ocenjevanju tveganja, ko želimo uporabiti AOP način programiranja.

3.3 Orodja pri uporabi tehnologije AOP

Kot smo že omenili, je AOP relativno nova tehnologija in zanjo obstaja več orodij, vendar ni vsako orodje zrelo za komercialno uporabo. Eden od glavnih dejavnikov za določanje zrelosti je sam uporabnik. Preden neko orodje štejemo za komercialno uporabno, moramo dobiti pozitivno povratno infor-

macijo od skupine aktivnih uporabnikov. Zaenkrat so komercialno uporabna naslednja orodja: AspectJ in AspectWerkz, ki sta se leta 2005 združili skupaj in sedaj imamo novo orodje AspectJ 5 [14], ter JBoss AOP in Spring AOP. Obstajajo še druga orodja, kot so abc, aspect#, AspectC++ in druga, ki zaenkrat niso v širši uporabi.
V nadaljevanju bomo pokazali en aspekt in kako bi vsako od vodilnih orodij ravnalo s tem aspektom. Ko govorimo o vodilnih orodjih, bomo opisali naslednja: AspectJ, AspectWerkz, JBoss in Spring AOP.

3.3.1 AspectJ 5

AspectJ 5 je, kot že omenjeno, nastal leta 2005, ko sta AspectJ in AspectWerkz s skupnimi močmi izdelala aspektno orientirano programsko platformo, pri čemer je vsak k projektu prispeval svoja strokovna znanja in tehnologije. V tem podpoglavju bomo najprej povedali, kakšno je bilo stanje, preden je prišlo do te združitve, in kakšno je zdaj.

AspectJ je razširitev AOP-ja, ki so ga ustvarili pri PARC-u za programski jezik Java. AspectJ je postal široko uporabljan " de facto " standard za AOP. Uporablja enako sintakso kot sam jezik Java in vključuje tudi IDE (angl. Integrated Development Environment) za prikaz strukture presečnih zadev.
Vsi veljavni programi Java so tudi veljavni programi AspectJ, vendar AspectJ omogoča, da programerji opredelijo aspekte. Lahko ga implementiramo na več različnih načinov, vključno z izvornim tkanjem (angl. source-weaving) ali bytecode tkanjem in neposredno v Javinem navideznem stroju (angl. Java Virtual Machine – JVM). Program AspectJ je veljaven Javin program, ki se izvaja v JVM. Razredi, na katere vplivajo aspekti, so binarno v skladu s tistimi, na katere ne vplivajo [15]. S podporo različnih implementacij je možna vzporedna rast s spreminjajočo se tehnologijo.
Prvotni Xerox AspectJ je uporabljal izvorno tkanje, ki zahteva dostop do izvorne kode. Ko je Xerox prispeval k razvoju programa Eclipse, je bil AspectJ

ponovno implementiran in začel je uporabljati Eclipsov prevajalnik ter bytecodeov tkalec, ki temelji na BCEL (angl. Byte Code Engineering Library), tako da lahko razvijalci pišejo kodo v binarni (.class) obliki.
V tem času je bil jezik AspectJ omejen na podporo razrednega modela, ki je bistven za load-time tkanje. IDE-integracija aspektov je postala hitrejša, kar je razvijalcem omogočilo izvajanje aspektov brez spreminjanja postopka grajenja. To je izboljšalo sprejem novega pristopa.
V nadaljevanju je prikazan aspekt Authentication, ki je napisan s pomočjo orodja AspectJ.

```
import banking.*;
public aspect Authentication {

        public pointcut authenticationRequired(Account account):
                execution(public * Account.*(..)) && this(account);

        before(Account account): authenticationRequired(account) {
                authenticate(account):
        }
```

Stični presek v zgornjem primeru uporabi modifikator in regularni izraz z maskirnim znakom *, da bi izrazil vse javne metode. Dostop do bančnega računa je mogoč prek parametra stičnega preseka. Nasvet uporabi ta parameter in stični presek ga veže na this(account). Posledično zajamemo objekt Account, kjer se metoda izvaja. Sicer pa je struktura nasveta podobna strukturi metode. Nasvet lahko vsebuje kodo za avtentikacijo, lahko pa, kot v tem primeru, pokliče neko drugo metodo [12].

AspectWerkz je dinamično, lahko in visoko performančno AOP-orodje, ki se uporabljalo v Javi. Prvič se pojavi leta 2002. Uporabljalo je bytecode-različico za tkanje našega projekta. To doseže s standardnim JVM-nivojskim API. Omogočalo je dodajanje, brisanje in ponovno strukturiranje nasvetov ter zamenjavo implementacije uvodov v času samega izvajanja programa. Če se vrnemo nazaj na naš primer, lahko opazimo, da je glavna razlika med AspectJ in AspectWerkz, da je Authentication zdaj navaden Javin razred,

torej ni aspekt. AspectWerkz, JBoss AOP in Spring AOP dodajo aspekte, ne da bi spremenili sintakse programskega jezika Java. AspectWerkz orodje je omogočalo dva načina deklaracije AOP-ja. Najpogosteje uporabljene so bile opombe (angl. annotations), ki so prikazane v nadaljevanju.

Orodje AspectWerkz je podpiralo tudi stil XML, ki je podoben tistemu, ki ga podpira JBoss AOP, ki ga bomo opisali v nadaljevanju. V dokumentu XML so posebej opisani aspekti.
Kot vidimo od zgoraj, je nasvet navadna deklaracija metode. Po dogovoru velja, da je to drugačen način deklaracije metode, ker se ne sklicuje eksplicitno, temveč deluje samodejno, ko je dosežen nek stiči presek. Stični preseki so v AspectWerkzu opredeljeni kot navadne string-vrednosti, ki kažejo na neko presečno metodo. Zato ni potreben mehanizem za uvoz stičnih presekov. Dostop do izvršujočega objekta Account je isti kot pri AspectJ.

Authentication.java

```
import org.codehaus.aspectwerkz.definition.Pointcut;
public class Authentication {
        @Expression("execution(public * banking.Account.*(..))")
        Pointcut authenticationRequired() {
                return null;
        }

        @Before("this(account) && authenticationRequired")
        public void before(Account account) throws Throwable {
                authenticate(account);
        }
```

aop.xml

```
<!DOCTYPE aspectwerkz PUBLIC "-//AspectWerkz//DTD//EN" "http//aspectwerkz.com">
<aspectwerkz>
        <system id="banking.example">
                <aspect class="banking.Authentication"/>
        </system>
</aspectwerkz>
```

aop.xml datoteka predstavi, kako so aspekti vključeni v sistemu.

AspectJ 5 omogoča razširitev orodja AspectJ, s tem da mu doda podporo za uporabo opombnega načina programiranja. Prav tako omogoča celotno AOP-podporo Java 5 ter razširi način load-time tkanja [16].

3.3.2 JBoss AOP

JBoss AOP se prvič pojavi leta 2004 in prestavlja 100 % čisto Javino aspektno usmerjeno ogrodje, ki se lahko uporabi v katerem koli programskem okolju. Zagotavlja čistejšo ločitev od aplikacijske logike in sistemske kode [17]. V kombinaciji s opombami JDK 1.5 je odličen način za razširitev programskega jezika Java. V nadaljevanju je prikazan primer enega aspekta v jeziku JBoss. Aspektni jezik JBoss uporablja XML-stil za predstavitev aspektov, presekov in navodil. Navodila kreiramo s pomočjo navadne javanske metode in jih nato pokličemo v datoteki XML [12].

Authentication.java

```
import org.jboss.aop.joinpoint.MethodInvocation;
import org.jboss.aspects.asynchronous.aspects.jboss.TraceThreadAspect;

public class Authentication extends TraceThreadAspect {

        public Object beforeAuth(MethodInvocation method) throws Throwable {
                authenticate((Account)method.getTargetObject());
                return method.invokeNext();
        }
```

jboss-aop.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>
        <aspect class="banking.Authentication" scope="PRE_VM" />
        <bind pointcut="execution(public void banking.Account-&gt;*(..))">
                <advice aspect="banking.Authentication" name=beforeAuth"/>
        </bind>
</aop>
```

V jboss-aop.xml datoteki definiramo našega preseka in naredimo povezavo z razredom Authentication.java. Nasvet pa napišemo v razredu Authentication.java.
Nekaj mesecev nazaj je prišla na trg nova različica JBoss aplikacijskega strežnika, ki se imenuje WildFly. Glede prejšnje verzije JBoss AS 7, nova verzija omogoča hitrejši zagon strežnika, rešuje probleme vezane z nalaganjem razredov (angl. classloading) in lažji način dodajanja novih uporabnikov, ki bodo upravljali ta strežnik. Isto tako omogoča podporo spletne vtičnice (angl. WebSocket) in ne-blokirane vhode/izhode (angl. Non-Blocking I/O), ki so značilne za Java EE 7.

3.3.3 Spring AOP

Spring AOP se je prvič pojavil leta 2004 kot dodatek k ogrodju Spring. Spring je enostavnejši za uporabo kot AspectJ, saj ni potrebe po posebnem postopku prevajanja. Trenutno podpira samo metodo izvajanja stične točke (join point). Operacija za prestrezanja polja (angl. field interception) zaenkrat ni implementirana, čeprav lahko dodamo podporo za to, ne da bi poškodovali jedra samih Spring AOP API-jev. Jezik Spring AOP se razlikuje od drugih jezikov, ki jih uporabljamo pri aspektnem programiranju. Cilj je, da se zagotovi tesna povezanost med samim Spring AOP-jem in Spring IoC-jem. V nadaljevanju je prikazan primer aspekta s pomočjo jezika Spring AOP.

Authentication.java

```
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class Authentication implements MethodBeforeAdvice {

        public void before(Method m, Object[] args,Object target)
                throws Throwable {
                authenticate((IAccount) target) ;
        }
```

springconfig.xml

```
<beans>
        <bean id="accountbean" class="org.springframework.aop.framework.ProxyFactoryBean">
                <property name="proxyInterfaces"> <!-- CONFIG -->
                        <value>banking.IAccount</value>
                </property>
                <property name="target">
                        <ref local="beanTarget"/>
                </property>
                <property name="interceptorNames">
                        <list><value>authenticationAdvisor</value></list>
                </property>
        </bean>
        <bean id="beanTarget" class="banking.Account"/> <!-- CLASS -->
        <bean id="authenticationAdvisor" class="org.springframework.aop.support/>
                <property name=advice"> <!-- ADVISOR -->
                        <ref local="authenticationBeforeAdvice"/>
                </property>
                <property name="pattern"><value>.*</value></property>
        </bean>  <!-- ADVICE -->
        <bean id="authenticationBeforeAdvice" class="banking.Authentication"/>
</beans>
```

Kot vidimo zdaj imamo več dela z XML dokumentom kot v zgornja dva primera. Podobno JBoss AOP-ju, Springov nasvet je isto Java metoda s posebnimi parametri, ki jih kličemo preko Springovega okvirja. V XML dokumentu imamo accountBean, ki daje Spring okviru dostop do Account objektov, vključno s prestreznikom (angl. interceptor), ki se uporablja za nasvet, svetovalec (angl. advisor), ki se ujema s svojim vzorcem in pred

svetovalcem uporaba tega vzorca.

Poglavje 4

Primerjava med aspektnim in objektnim razvojem programske opreme

Veliko ljudi misli, da je aspektni način razvijanja programske opreme nekaj, kar je v nasprotju z že obstoječim objektnim načinom programiranja. Vendar to ni res. Aspektni način programiranja nekako dopolnjuje že obstoječi objektni način programiranja, tako da omogoča razvijalcu, da dinamično spreminja statično objektno usmerjen model, da bi naredil sistem, ki se lahko prilagodi novim zahtevam. Torej, na aspektni način ne gledamo kot na neko primerjavo, temveč kot na neko dopolnitev objektnega pristopa.
Aspektni način programiranja je neka vrsta " metaprogramiranja ". Vse, kar se da narediti s pomočjo aspektnega načina, je mogoče narediti tudi brez njega, le da moramo v tem primeru dodati več kode. Aspektni način le prihrani pisanje te dodatne kode.
V nadaljevanju je opisan program, ki je najprej napisan s pomočjo objektnega pristopa, nato pa mu sledi izboljšava s pomočjo aspektnega načina. Aspektni način je napisan v AspectJ. Za uporabo AspectJ-ja smo uporabili AJDT (angl. AspectJ Development Tools), ki predstavlja dodatek k orodju Eclipse.

Program zapisan brez uporabe aspektov v programskem jeziku Java.

```
import java.security.AccessController;

public class Racun {
    private int stevilkaRacuna;
    private float stanje;

    public Racun(int stevilkaRacuna) {
        this.stevilkaRacuna = stevilkaRacuna;
    }

    public int getStevilkaRacuna() {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
        return stevilkaRacuna;
    }

    public void credit(float znesek) {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
        stanje = stanje + znesek;
    }

    public void debit(float znesek) throws InsufficientBalanceException {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
        if (stanje < znesek) {
            throw new InsufficientBalanceException("Insufficient total balance");
        } else {
            stanje = stanje - znesek;
        }
    }

    public float getStanje() {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
        return stanje;
    }

    public String toString() {
        return "Racun: " + stevilkaRacuna;
    }
}
```

Kot vidimo v zgornjem primeru, se vrstica

```
AccessController.checkPermission(
        new BankingPermission("accountOperation"));
```

večkrat ponavlja. To morda v zgornjem primeru ne predstavlja neke večje težave, vendar če klicemo to metodo, recimo, več kot trikrat, lahko to postane zelo moteče.
Zdaj bomo pokazali, kako se tega lotimo na zelo lep in eleganten način. V nadaljevanju je opisan aspektni način oziroma napisan aspekt, ki nekako izboljša objektni način programiranja.

Zapis aspekta.

```
private static aspect PermissionCheckAspect {

    private pointcut permissionCheckedExecution() :
        (execution(public int Racun.getStevilkaRacuna())
         || execution(public void Racun.credit(float))
         || execution(public void Racun.debit(float)
                              throws InsufficientBalanceException)
         || execution(public float Racun.getStanje()))
        && within(Racun);

    before() : permissionCheckedExecution() {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
    }
}
```

Kot vidimo iz zgornjega primera, najprej definiramo en presek, v našem primeru je to permissionCheckedExecution, ki vzame stične točke getStevilkoRacuna(), credit(), debit() ter getStanje(). Potrebujemo še navodilo before, da bi program naredili preglednejši in si s tem prihranili čas.
Obstaja pa še lepši zapis zgornjega aspekta, ki je še krajši in še preglednejši.

Zapis aspekta z uporabo regularnih izrazov.

```
private static aspect PermissionCheckAspect {
```

```
    private pointcut permissionCheckedExecution()
        : (execution(public * Racun.*(..))
           && !execution(String Racun.toString()))
          && within(Racun);

    before() : permissionCheckedExecution() {
        AccessController.checkPermission(
            new BankingPermission("accountOperation"));
    }
}
```

Kot vidimo iz zgornjega primera, aspektni način nekako nadgradi že obstoječi objektni način in ga naredi še preglednejšega. Uporabili smo regularni izraz z maskiranim znakom *, kako bi skrajšali naš program. Z zgornjim izrazom public * Racun.*(..), povemo, da bomo vzeli vse metode razreda Racun in pri tem, da ni pomemben tip, ki ga metode vračajo. Torej, če imamo veliko metod, ki se prekrivajo, je boljše, da uporabimo aspektni način programiranja, saj s tem prihranimo veliko časa in kode. Za programe, kjer imamo zelo malo metod, aspektni način morda ni najprimernejši.

AOP je odlična značilnost, ki se mora uporabljati zelo previdno. Ne smemo pomešati navadne kode in aspektne kode pri enem in istem aspektu, ker v tem primeru ne bomo imeli korist iz aspektnega načina in bomo porušili hierarhijo aspektnega načina. Lahko na koncu povemo še da je testiranje programa, ki je napisan s pomočjo objektnega načina, lažji kot z uporabo aspektnega načina, ker pri aspektnem načinu ne moremo zaporedno prebrati celotno kodo. Bralec mora podrobno preučiti in razumeti aspekte in njihova presečišča.

Poglavje 5

Sklepne ugotovitve

V okviru diplomskega dela smo si ogledali razvoj metodologij. Najprej smo ugotovili, da so imeli ljudje brez teh metodologij veliko težav pri razvoju programske opreme in je pogosto prihajalo do zamud pri izvajanju projektov. Nato smo si na kratko ogledali, kako so se metodologije razvijale skozi čas. Podrobneje smo opisali objektni pristop razvoja programske opreme. Sledil je aspektni razvoj programske opreme.
Primarni cilj diplomske naloge je bil predstaviti aspektni pristop pri razvijanju opreme. Aspektni način razvoja programske opreme je uvedel nov pristop pri obravnavanju presečnih zadev, ki se pojavijo kot rezultat zapletenosti in razmetanosti kode. Do pojava tega pristopa so imeli tradicionalni pristopi veliko težav s presečnimi zadevami. V nadaljevanju smo si ogledali tehnike za odpravo teh težav. Opisali smo tudi najbolj uporabljan jezik za aspektni razvoj programske opreme. Na koncu smo naredili še primerjavo med trenutno najbolj uporabljanim pristopom in aspektnim pristopom. Ugotovili smo, da aspektni pristop ni nek revolucionarni pristop, temveč nekako nadgradi že obstoječi objektni pristop in pomaga odpraviti težave, ki jih imamo pri objektnem pristopu. Zaenkrat lahko povemo še, da aspektni način ni zelo razširjen v praksi, saj je še vedno v fazi testiranja in dokumentacije, pričakuje pa se, da bo v prihodnosti postal eden izmed najbolj uporabljanih načinov za razvoj programske opreme.

Slike

Literatura

[1] A. Turing, "Computable numbers, with an application to the decision problem". Dostopno na: http://www.davebollinger.com/about.htm (zadnji obisk: maj 2013)

[2] Structured programming. Dostopno na: http://en.wikipedia.org/wiki/Structured_programming (zadnji obisk: maj 2013)

[3] Software Development Methodologies. Dostopno na: http://www.codeproject.com/Articles/124732/Software-Development-Methodologies (zadnji obisk: maj 2013)

[4] G. Fitzgerald, D. Avison, Information system development methodologies,techniques & tools, London [etc.] : McGraw-Hill, 2008, cop. 2006

[5] Software Development Life-Cycle. Dostopno na: http://en.wikipedia.org/wiki/Software_development_process (zadnji obisk: junij 2013)

[6] Enotna metodologija razvoja programske opreme. Dostopno na: http://www2.gov.si/mju/emris.nsf/Zvezek1?OpenFrameSet (zadnji obisk: junij 2013)

[7] Objektno usmerjen razvoj programske opreme. Dostopno na: http://www.cek.ef.uni-lj.si/magister/slokar546.pdf

[8] Aspektno usmerjen razvoj programske opreme. Dostopno na: lgm.fri.uni-lj.si/PA/ASPEKT_USM_RAZVOJ/aspekt_usm_razvoj.ppt (zadnji obisk: julij 2013)

[9] Primer presečnih zadev. Dostopno na: http://oberon2005.oberoncore.ru/paper/np2002.pdf (zadnji obisk: junij 2013)

[10] Aspect oriented programming. Dostopno na: http://en.wikipedia.org/wiki/Aspect-oriented_programming (zadnji obisk: junij 2013)

[11] Ian Gorton, Essential Software Architecture, Second Edition. Springer(2010)

[12] AOP tools comparison. Dostopno na : http://www.ibm.com/developerworks/library/jaopwork1/#figure1 (zadnji obisk: avgust 2013)

[13] Primerjava med aspektnim in objektnim načinom programiranja. Dostopno na: http://stackoverflow.com/questions/232884/aspectoriented-programmingvsobjectorientedprogramming (zadnji obisk: avgust 2013)

[14] AspectJ 5. Dostopno na : http://aspectwerkz.codehaus.org/index-merge.html (zadnji obisk: avgust 2013)

[15] AspectJ. Dostopno na : http://en.wikipedia.org/wiki/AspectJ (zadnji obisk: avgust 2013)

[16] AspectJ 5 novosti. Dostopno na : http://eclipse.org/aspectj/doc/released/faq.php#q:aspectj5features (zadnji obisk: avgust 2013)

[17] JBoss AOP. Dostopno na :
http://www.jboss.org/jbossaop (zadnji obisk: avgust 2013)

[18] Spring AOP. Dostopno na :
http://static.springsource.org/spring/docs/2.0.x/reference/aop.html
(zadnji obisk: avgust 2013)

[19] Slapovni model. Dostopno na:
http://en.wikipedia.org/wiki/Waterfall_model (zadnji obisk: september 2013)

[20] Primer tkanja . Dostopno na
http://en.wikipedia.org/wiki/Aspect_weaver (zadnji obisk: september 2013)

[21] Jacobsen, I. and Ng, P-W, Aspect-oriented Software Development with Use Cases, Boston: Addison-Wesley (2004)

[22] Clark, S. and Baniassad, E., Aspect-Oriented Analysis and Design: The Theme Approach, Harlow, UK: Addison-Wesley (2005)

[23] Kotonya, G. and Sommerville, " Requirements engineering with viewpoints ", BCS/IEE Software Eng. J (1996)

[24] Aspektno usmerjeno modeliranje in načrtovanje. Dostopno na:
http://ifs.host.cs.st-andrews.ac.uk/Books/SE9/SampleChapters/PDF/Chap21-AOSD.pdf(zadnji obisk: september 2013)